MALIBOO: When Machine Learning meets Bayesian Optimization

Bruno Guindani
Department of Electronics, Information and Bioengineering
Politecnico di Milano
Milano, Italy
bruno.guindani@polimi.it

Danilo Ardagna
Department of Electronics, Information and Bioengineering
Politecnico di Milano
Milano, Italy
danilo.ardagna@polimi.it

Alessandra Guglielmi
Department of Mathematics
Politecnico di Milano
Milano, Italy
alessandra.guglielmi@polimi.it

Abstract—**Bayesian Optimization (BO) is an efficient method for finding optimal cloud computing configurations for several types of applications. On the other hand, Machine Learning (ML) methods can provide useful knowledge about the application at hand thanks to their predicting capabilities. In this paper, we propose a hybrid algorithm that is based on BO and integrates elements from ML techniques, to find the optimal configuration of time-constrained recurring jobs executed in cloud environments. The algorithm is tested by considering edge computing and Apache Spark big data applications. The results we achieve show that this algorithm reduces the amount of unfeasible executions up to 2-3 times with respect to state-of-the-art techniques.**

Index Terms—**Cloud computing, Bayesian optimization, Black-box optimization, Machine learning**

I. Introduction

Cloud computing environments are widely employed in several industries since they allow to easily adjust the allocated resources (CPU, memory, disk, network, etc) to match the needs of the software application of interest. However, the configuration of such environments is in the charge of the cloud end user. A poor choice of software and/or virtual hardware settings can lead to huge additional costs for them, and such costs could have been prevented if a more suitable configuration had been chosen [1]–[3]. For instance, the average configuration costs of big data applications can be more than three times as much as the optimal one, and up to 10 times in the worst case [1]. This is especially true for recurring cloud jobs, i.e., applications that need to be executed multiple times, where additional cost of suboptimal configurations piles up over time.

It is also of paramount importance that the execution of a cloud application complies with given time deadlines, either coming from the cloud provider as part of its business model (e.g., providing several performance targets/Service Level Objectives), or by the cloud end users' needs. Configuration of cloud jobs can heavily impact their running time [2]–[4], for better or for worse, and a poor choice of parameters may lead to an out-of-time execution. However, as a consequence of the diverse behavior and resource requirements of cloud jobs, choosing the best configuration for a broad spectrum of applications is a challenging task.

Bayesian Optimization (BO) has recently gained notoriety as a powerful tool to solve global optimization problems in which expensive, black-box functions are involved [1], [3], [5]–[10]. BO is a sequential design strategy that requires few steps to get sufficiently close to the true optimum, while requiring no derivative information on the optimized function. Most commonly, it is initialized by choosing and evaluating a small handful of starting points, then fitting a Gaussian Process model using these points. The fitted model provides an estimate of both the function value at each point and the uncertainty around the estimate. BO then iteratively chooses new points at which to evaluate the function in a such a way to balance exploration (large uncertainty) and exploitation (large expected value) [5], [8].

Machine Learning (ML) models are another popular tool in the world of Information and Communications Technology (ICT) systems. Their remarkable predicting capabilities can assist the user in accurately predicting resource usage, execution times, etc. Indeed, previous work [11]–[17] has shown that ML models are usually able to predict these target quantities with very small validation error.

This paper proposes MALIBOO (MAchine Learning In Bayesian OptimizatiOn), a tool integrating BO algorithms with ML techniques with the goal to minimize the execution costs of recurring time-constrained cloud computing jobs.

Our approach is validated in a number of scenarios of interest, including edge computing applications and big data analytics. The proposed BO algorithm variants reduce both the unfeasible executions and the percentage of unfeasible costs up to 2-3 times with respect to state-of-the art techniques.

This paper is organized as follows. Section II surveys the state of the art for both BO and ML techniques, discussing the fields they have been applied to. In Section III, we formalize the problem of optimal cloud configuration at hand. Section IV presents an overview of BO and explains its key components. Our contributions involving the integration of ML into BO are detailed in Section V. Finally, we present results for experimental validation of our proposed algorithm in Section VI,

The European Commission has partially funded this work under the Horizon 2020 Grant Agreement number 956137 LIGATE: LIgand Generator and portable drug discovery platform AT Exascale, as part of the European High-Performance Computing (EuroHPC) Joint Undertaking program.

and some conclusive comments follow in Section VII.

II. Related work

In this work, we introduce a new algorithm to find optimal cloud configurations when job execution is subject to time constraints. Our main contribution is integrating information coming from ML models in an algorithm based on BO. In this section, we review the state of the art for these techniques, with particular focus on their application to ICT systems in a cloud computing setting. In particular, we examine BO literature in Section II-A and ML literature in Section II-B.

A. Bayesian Optimization

This work builds on previous results found in [1], in which the *CherryPick* system is applied to optimize the execution of Apache Spark big data applications in cloud computing systems. *CherryPick* exploits pure constrained BO to find optimal cloud configurations.
Another instance of BO being applied to cloud computing is the RAMBO framework presented in [18]. The authors are interested in cloud applications composed of hundreds of microservices, and they use multi-objective BO to obtain optimal resource allocation for these microservices. On the other hand, [19] introduces the IMGPO algorithm, which uses hierarchical domain partitioning to select points for BO to sample. The PROTOCOL algorithm is proposed in [20] as a parallel extension to IMGPO, and it is used to find optimal settings for the equipment used in experimental protocols taking place in cloud laboratories. Finally, [9] presents Google Vizier, a Google-internal framework to conduct black-box optimization of physical and software systems on the company's servers. Their default optimization algorithm exploits BO based on Batched Gaussian Process Bandits.

In general, BO has been widely studied for a number of black-box optimization problems [3], [5], [8], [10], ranging from energy minimization for molecule simulation [21] to laser-plasma accelerated electron beams [22]. The topic of constrained BO also has received attention, including works on unknown and/or black-box constraints [6], [23]–[25].

B. Machine Learning

Our work is motivated by the belief that ML techniques can lend their predicting capabilities to BO to further improve its effectiveness. ML has been widely applied to predict the performance of ICT systems, such as big data applications, training of deep learning models, and Functions as a Service (FaaS) systems. Overall, results found in literature are promising in showing the usefulness of ML in the context of cloud computing configuration. For instance, [12] examines the performance of several ML models in carrying out predictions of execution times of Apache Spark cloud jobs with different types of workloads. Their results outperform models used by the original creators of Spark. Furthermore, [15] proposes a ML-based prediction platform for Spark SQL queries and ML applications, which exploits features related to each stage of the Spark application, as well as previous knowledge of the application profile. The design of the Hemingway framework [16] embeds the Ernest model and is specialized in the modeling and identification of optimal cluster configuration for Spark MLlib based applications. Authors of [13] employ several ML models alongside anomaly detection to properly configure a cloud-based Internet of Things (IoT) device manager while respecting Quality of Service (QoS) constraints. Another recent work, [11] explores performance prediction of training times of GPU-deployed neural networks starting from software and hardware cloud configuration specifications, by using ML techniques and feature selection methods. Similarly, [14] compares some popular ML techniques applied to a workload prediction analysis on HTTP servers, showing that these techniques all achieve good predicting capabilities. [17] uses ML models to observe and predict the evolution over time of the performance of systems at the Infrastructure as a Service (IaaS) cloud level. Finally, the Schedulix framework proposed in [26] uses linear regression to estimate execution latencies of serverless application in a public cloud FaaS setting.

Recent ML works involving BO include [27]–[30]. In particular, [27] presents the Paprika scheduler, which is able to co-optimize hardware and software configurations for Spark workloads. This process uses a BO-based algorithm integrated with ML elements, namely feature selection via Lasso regression. Their model is one of the closest ones to our research quest, but it requires offline training before deployment, which may be not always available, especially for recently released software. Furthermore, its exploitation of ML is arguably limited in scope, since it is only used to choose among the existing features without any regression strategy being involved. Authors of [28] and [29] propose the SVM-CBO algorithm exploiting ML to find optimal configurations in a constrained setting, for instance when deploying Deep Neural Networks to tiny, micro-controller based systems. Their algorithm consists of two phases. The first phase approximates the feasible domain via a Support Vector Machine regression model, while in the second phase, pure BO is applied to the approximated domain found by ML. This approach is arguably inefficient in exploiting the iteration budget, since it handles separately the domain approximation and optimization phases. Finally, [30] introduces the Lynceus framework to optimize data analysis and ML jobs on cloud platforms that are constrained by time deadlines. It exploits BO using an unconventional Decision Tree bagging ensemble as its prior distribution to model probability of respecting the constraints.

III. Problem overview

Our goal is to find the optimal cloud configuration for running an application in a cloud cluster, while abiding by a given time deadline. We are particularly interested in recurring applications, which constitute up to 40% of the total amount of analytic jobs [1], [31]–[33]. Picking a good configuration for a recurring job holds utmost importance, because the total extra costs caused by a poor configuration accumulate over time. The recurring setting also allows to amortize the cost of a tuning campaign, since the up-front cost is outweighed by the savings in the long run.

We consider the mathematical formulation for our constrained, noisy global optimization problem similarly to [1]. Let $x \in \mathcal{A}$ denote the d-dimensional vector representing a configuration for the cloud job, with $\mathcal{A} \subset \mathbb{R}^d$ being the domain of all possible configurations. The vector x can include, e.g., the buffer size, the memory type (SSD vs HDD), other application-specific parameters (algorithm iterations, number of simulated molecules), and, in particular, the number of total cores (possibly available on multiple homogeneous VMs) used for the job. The *objective function* to be minimized is the total cost $f(x) = P(x)T(x)$, where $T(x)$ is the unknown execution time and $P(x)$ is a known, deterministic function representing the price per time unit of configuration x. We also assume the constraint $T(x) \leq T_{max}$, where T_{max} is a given time threshold. Hence the problem is to find:

$$\begin{aligned} &\min_{x\in\mathcal{A}} f(x) = P(x)T(x) + \varepsilon \\ &\text{with} \quad \varepsilon \sim \mathcal{N}(0, \eta^2) \\ &\text{s.t.} \quad T(x) \leq T_{max}, \end{aligned} \tag{1}$$

where ε is a noise term with unknown variance η^2, which is estimated by log-marginal-likelihood maximization. In an ICT setting, the noise term proves necessary in order to account for the intrinsic variability of an application execution time, even when it is run multiple times with the same configuration. This variability is typically produced by external causes such as access to underlying physical resources or network congestion. In this paper, we assume the deterministic price function $P(x)$ as being proportional to the number of virtual machines or cores used by the application job, which is always included in the cloud configuration vector x. Other choices of the price function are possible.

IV. Bayesian optimization background

Within the setting described in Section III, BO is an efficient method because it approximates the minimum of a given black-box objective function $f(\cdot)$ by using as few iterations as possible. Namely, we want to find $\widehat{x}$ where $\widehat{x} = \arg\min_{x\in\mathcal{A}} f(x)$. Strong assumptions on $f(\cdot)$ or on the minimization domain $\mathcal{A}$ are not required, and BO algorithms are derivative-free. For these reasons, BO is often used to optimize expensive black-box objective functions [1], [3], [5]–[10], that is, functions for which little to no information is available, and whose evaluation has significant time, resource, and/or monetary costs. This is exactly the situation at hand with cloud performance optimization, where running many experiments in an attempt to find the optimal configuration would defeat the main purpose of decreasing overall costs for applications maintenance and execution, especially for recurring jobs.

We now give a brief overview on the main tools used by BO. In particular, we shall explain the peculiarities of *Gaussian processes*, *posterior distributions* (Section IV-A), and *acquisition functions* (Section IV-B) in a BO context. Lastly, we summarize this overview in Section IV-C.

A. Prior and posterior distributions, Gaussian Process

The core idea of BO comes from the Bayesian approach to statistics, in which values taken by $f(\cdot)$ are treated as random variables, and a *prior distribution* represents the a-priori information on the modeled phenomenon – in the case of BO, information on the location of the minimum of $f(\cdot)$. The prior distribution is then iteratively updated with information coming from the observed data, obtaining the *posterior distribution*.

In this context, the *Gaussian process* (GP) [34] is the preferred choice for the prior for $f(\cdot)$. For any $x \in \mathcal{A}$, this prior assigns to the value of $f(x)$ a Gaussian probability distribution which depends on x:

$$f(x) \sim \pi_x(\cdot) = \mathcal{N}(\mu_0(x), \sigma_0^2(x, x)). \tag{2}$$

Functions $\mu_0(\cdot)$ and $\sigma_0^2(\cdot, \cdot)$ are called mean and kernel functions, respectively, and they are the GP model hyperparameters. These functions serve as the "initial guesses" on values of $f(\cdot)$ and its uncertainty, from which the BO algorithm starts and which will later be updated with observed values. A constant mean function $\mu_0(\cdot) \equiv \mu_0$ is often adopted, whereas the choice of the kernel is more delicate, since it influences the smoothness of the process. Commonly used kernels include the squared exponential or Radial Basis Function and the Matérn kernel [34]. The former gives the GP an excessively large degree of smoothness which is unrealistic in many practical scenarios. Therefore, in this work we assume $\mu_0(\cdot) \equiv \mu_0$ and we use the Matérn kernel [5] with smoothness parameter ν:

$$\sigma_0^2(x, x') := \frac{1}{2^{\nu-1}\Gamma(\nu)} \left(\sqrt{2\nu}\|x - x'\|\right)^{\nu} K_\nu \left(\sqrt{2\nu}\|x - x'\|\right),$$

where $\Gamma(\cdot)$ and $K_\nu(\cdot)$ are the gamma function and the modified Bessel function [35], respectively.

We now examine the posterior distributions for these hyperparameters. Let $H = \{(x_1, f(x_1)), \ldots, (x_n, f(x_n))\}$ be the history of n past observations, which we also indicate as H_n when emphasizing its cardinality. Specifically, observation i consists of the configuration vector x_i and the associated evaluation of the objective function $f(x_i)$. Having observed values in H, one can compute the posterior distribution of each $f(x)$, starting from the prior distribution in Eq. (2) and taking these observations into account. The posterior is the conditional distribution of $f(x)$ given H_n, which, in this case, is a Gaussian distribution with mean $\mu_n(\cdot)$ and variance $\sigma_n^2(\cdot)$:

$$f(x)|H_n \sim \pi_x(\cdot|H_n) = \mathcal{N}(\mu_n(x), \sigma_n^2(x)). \tag{3}$$

Note the conditioning symbol $|$ in Eq. (3). The posterior mean and variance can be computed in closed form by well-known properties of GPs [5]:

$$\begin{aligned} \mu_n(x) = \mu_0(x) + \sigma_0^2(x, x_{1:n})^T \, \sigma_0^2(x_{1:n}, x_{1:n})^{-1} \cdot \\ \cdot \left(f(x_{1:n}) - \mu_0(x_{1:n})\right), \end{aligned} \tag{4}$$

$$\begin{aligned} \sigma_n^2(x) = \sigma_0^2(x, x) - \sigma_0^2(x, x_{1:n})^T \, \sigma_0^2(x_{1:n}, x_{1:n})^{-1} \cdot \\ \cdot \, \sigma_0^2(x, x_{1:n}). \end{aligned} \tag{5}$$

In Eqq. (4) and (5), $\sigma_0^2(x, x_{1:n})$ indicates the column vector of values of the $\sigma_0^2(\cdot,\cdot)$ function applied to pairs $(x, x_1), \dots, (x, x_n)$, and similarly for $f(x_{1:n})$ and $\mu_0(x_{1:n})$. Analogously, $\sigma_0^2(x_{1:n}, x_{1:n})$ is the matrix of values of $\sigma_0^2(x_i, x_j)$ with $i, j = 1, \dots, n$.

Given a configuration x, after n evaluations, this probabilistic model allows us to attribute both the current pointwise estimate of $f(x)$ and a measure of uncertainty on such estimate, represented by $\mu_n(x)$ and $\sigma_n^2(x)$ respectively.

B. Acquisition function

In the previous section, we described how BO models the objective function at iteration n. We now move on to explain how BO chooses the next iteration point x_{n+1} based on such model.
At each step, BO formulates a proxy problem – the maximization of the *acquisition function* $g(x)$, which depends on the GP model at the current algorithm iteration and measures the utility of evaluating the objective function $f(x)$ in a given configuration x. This function is optimized at each round of the iterative algorithm, instead of directly optimizing the objective function itself. In fact, the acquisition function is available in closed form and it is inexpensive to evaluate. Therefore, fast heuristics are available to solve this proxy problem, even using derivative information about the acquisition function [5].
Being a measure of utility, the acquisition function takes on larger values in points in which the algorithm should choose to evaluate the objective function, in order to get the most information about the location of the optimum. This means that the acquisition function must strike a delicate balance – the exploration-exploitation trade-off. On one side, we have points to which large uncertainty is attached, for instance because they lie in a region of the domain which has not been explored yet. Choosing such points to evaluate the objective $f(\cdot)$ is appealing, especially early on in the optimization procedure, because this would allow a massive decrease of the overall uncertainty, i.e., an increase of the amount of available information about the optimum. On the other hand, the algorithm does seek to find the optimum of the objective function, therefore it should also choose to evaluate points which most likely (according to the GP model) give small values of $f(\cdot)$. This is done by exploiting the information already available on the location of the optimum, especially in the late iterations of the algorithm.

In this paper, we consider the Expected Improvement acquisition function and its constrained extensions. The *Expected Improvement* (EI) over the best value f_n^* found by the optimization process so far is:

$$\begin{aligned} EI_n(x) &:= \mathbb{E}_{\pi_x(\cdot|H_n)}[\max(f_n^* - f(x), 0)] \\ \text{with } \ f_n^* &= \min_{i \le n} f(x_i). \end{aligned} \tag{6}$$

The expectation is taken under the current posterior distribution $\pi(\,\cdot\,|H_n)$ of $f(x)$, given history H_n. Eq. (6) means that we maximize the expected value of the improvement over the current best point f_n^*, based on the information collected so far (i.e., the points in the history H_n). We consider a generalization of EI to the constrained optimization setting – the *Expected Improvement with Constraints* (EIC) acquisition function [7], which accounts for the probability of a point respecting the constraints:

$$\begin{aligned} EIC_n(x) &:= EI_n(x) \cdot \mathbb{P}_{\pi_x(\cdot|H_n)}\big(T(x) \le T_{max}\big) \\ &= EI_n(x) \cdot \mathbb{P}_{\pi_x(\cdot|H_n)}\big(f(x) \le P(x)\, T_{max}\big), \end{aligned} \tag{7}$$

In the last equality, we write the constraint $T(x) \le T_{max}$ as a function of $f(x) = P(x)T(x)$ since the GP prior is placed on $f(\cdot)$, not on $T(\cdot)$. The EIC in Eq. (7) is the same acquisition function used in *CherryPick* [1].

C. Summary of Bayesian Optimization

The procedure for a BO algorithm is summarized in Algorithm 1.

Algorithm 1 Generic Bayesian Optimization algorithm

```
1: choose $n_0$ initial points
2: evaluate $f(\cdot)$ in the initial points, add all evaluations to history $H$
3: for iterations $n = 1 : N$ do
4:     update the current posterior distribution of the GP model with data in $H$
5:     find point $x_{n+1}$ which maximizes the acquisition function $g(\cdot)$ under the current model
6:     evaluate $f(x_{n+1})$, add performed evaluation to $H$
7: end for
8: return estimated optimum $\widehat{x}$
```

In step 1, a small number n_0 of initial points are selected, usually 3 to 10, in order to initialize the algorithm. These points should be chosen so as to cover the maximum domain area possible, for instance using a Latin hypercube design [36], which maximizes the distance among those points. We then evaluate these points, i.e., we run the application with the given initial configurations, recording the points and their evaluations (step 2). After that, we enter the algorithm loop, where we update the posterior distribution (step 4), choose the next point x_{n+1} by maximizing the acquisition function (step 5), and evaluate it (step 6). The algorithm stops when the iteration budget N runs out.

Fig. 1 presents a visual summary on how BO works. In the top panel, the solid gray line represents the true objective function $f(\cdot)$ to be minimized. We also plot the GP estimates produced by $f(\cdot)$, in terms of its mean function (dashed blue line) and 95% confidence intervals (light blue area). The three red dots represent sampled points, which have smaller uncertainties attached to them with respect to all other domain points (note that utility in such points is still greater than zero if accounting for noise in observations of $f(\cdot)$). As previously explained (see Section IV-A), the GP model attaches a Gaussian probabilistic estimate to $f(x)$, for each x. In the top panel of Fig. 1, the Gaussian curve in light grey represents this estimate when $x = 2.51$. In the bottom panel of the figure, we plot instead the values of the acquisition

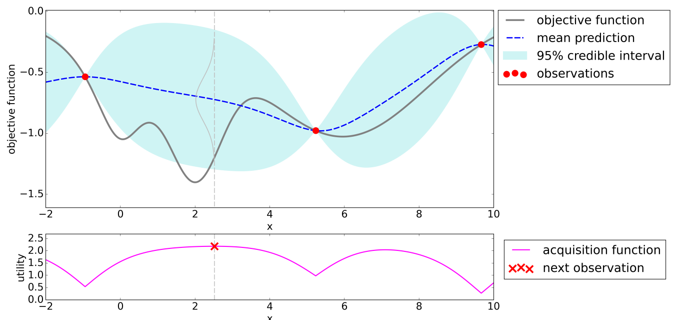

Fig. 1. Bayesian Optimization after 3 iterations.

function for each value of x. The crossed red point is the one which has been selected to be evaluated in the next round, since it has the largest expected utility among all points in the domain.

V. Encoding information from Machine Learning in Bayesian Optimization

Our main contribution is the integration of ML techniques into the BO framework. ML methods can prove to be useful additions to optimization problems because of their predicting capabilities. This is especially true in a setting where evaluating the objective function $f(\cdot)$ is expensive, and therefore a cheap estimation of values of $f(\cdot)$ can make up for the scarcity of direct information on them. In our case, it can be used to convey information about the violation of constraints, by guiding the search towards points $x \in \mathcal{A}$ which most likely will respect constraint limits. This is appreciated by the BO algorithm in the cloud configuration setting. Indeed, our goal is ultimately to find optimal (or near-optimal) configurations which are also feasible, i.e., points x with $T(x) \leq T_{max}$; unfeasible points represent a waste of resources and computational time in a recurring job setting, providing additional unnecessary costs.
The use of ML models is also motivated by their great accuracy in predicting execution times of the applications at hand. Besides the promising results on the matter in the literature [11]–[17], [26], our preliminary analysis shows good prediction capabilities on the target applications, as we shall see in Section VI-B. The idea of BO incorporating information which is independent of its GP model was first considered in [37].

A. Novel acqusition functions

The core contribution of MALIBOO is integrating the acquisition function (described in Section IV-B) with information coming from ML models. Let $\widehat{T}(x)$ be a generic predicting function for $T(x)$, that is, a function which outputs a prediction on the execution time of a cloud job given configuration x. In our case, $\widehat{T}(\cdot)$ is a ML regression model trained on all data collected so far by the BO algorithm, as explained in the previous sections. First and foremost, note that using a model that predicts execution time given configuration x is equivalent to using one predicting the total cost, since they only differ by a known, deterministic constant, i.e., the configuration price $P(x)$ (see Eq. (1)).
We propose several novel acquisition functions which incorporate the information coming from $\widehat{T}(\cdot)$ into the BO algorithm:

A) $g_A(x) = EIC(x)$: the original Expected Improvement with Constraints acquisition function, used by *CherryPick* [1], which we use as baseline;
B) $g_B(x) = g_A(x) \cdot \exp(-k\,\widehat{T}(x))$, the latter term being a $[0,1]$-valued weight for g_A, and called a nascent minima distribution function [38]. This function serves the purpose of turning $\widehat{T}(x)$ into an acquisition-like function, that is, giving values closer to 1 the smaller the predicted execution time $\widehat{T}(x)$ is (therefore being a desirable point), and closer to 0 if such prediction is large;
C) $g_C(x) = g_A(x) \cdot I_{\{\widehat{T}(x) \leq T_{max}\}}$, with I being the indicator function: the acquisition function is set to zero, i.e., the search is outright prevented, in areas where the predicted execution time violates the threshold T_{max}, at least in the current algorithm iteration. Basically, we use the model $\widehat{T}(x)$ to approximate the feasible domain at the current iteration;
D) $g_D(x) = g_A(x) \cdot \exp(-k\,\widehat{T}(x)) \cdot I_{\{\widehat{T}(x) \leq T_{max}\}}$: the combination of cases B and C.

Note that the original definition of the nascent minima distribution functions (used in variants B and D) includes the normalization constant $1/C_k$ as a multiplicative factor, with $C_k = \int_{\mathcal{A}} \exp(-k\,\widehat{T}(w))\,\mathrm{d}w > 0$, as described in [38]. However, this value is independent of x, therefore we can omit it when maximizing the acquisition function in x.

B. Proposed algorithm

We summarize the complete procedure in Algorithm 2. MALIBOO is based on BO (see Algorithm 1), but it integrates

Algorithm 2 MALIBOO algorithm

```
1:  choose $n_0$ initial points
2:  evaluate $f(\cdot)$ in the initial points, add all evaluations to history $H$
3:  for iterations $n = 1 : N$ do
4:      update the current posterior distribution of the GP model with data in $H$
5:      if If $g(\cdot) \neq g_A(\cdot)$ then
6:          train model $\widehat{T}(\cdot)$ with data in $H$, to be used in $g(\cdot)$
7:      end if
8:      find point $x_{n+1}$ which maximizes the acquisition function $g(\cdot)$ under the current model
9:      evaluate $f(x_{n+1})$, add performed evaluation to $H$
10:     update memory queue with $x_{n+1}$
11:     if stopping criteria are met then
12:         terminate the algorithm
13:     end if
14: end for
15: return estimated optimum $\widehat{x} = \arg\min_{x \in H} f(x)$
```

elements from ML techniques. We use a first-in-first-out memory queue for discrete features to prevent exploration of already visited values, inspired by the taboo search meta-heuristic methods [39]. In this memory queue, we save the last q configuration vectors visited by the algorithm. Configurations currently in the queue are excluded from being selected again until they have shifted out of the queue, i.e., after q iterations.

Similarly to regular BO, at each round, we maximize (see Algorithm 2, step 8) one of the acquisition functions presented in Section V-A, which incorporates the ML model trained at step 6. Then, we evaluate the newly chosen configuration (step 9) as usual, and we update the aforementioned memory queue (step 10). The algorithm continues until the evaluated running time at the current iteration is sufficiently close to the time threshold: $T(x_n) \in [\alpha T_{max},\ T_{max}]$, with $\alpha \in (0, 1)$ (step 12). Our goal is to obtain a configuration that is compliant with the time threshold, but also uses as few resources as possible. Generally speaking, using more resources results in a lower execution time – meaning that a time which is just under the threshold likely consumes the least amount of resources for that configuration to be feasible. After termination, it is likely that we have found the true optimal configuration because of the convergence properties of the BO algorithm. Afterwards, we execute subsequent runs using such optimal or near-optimal configuration.

VI. Experimental results

In this section, we present experimental results for validating MALIBOO. In Section VI-A, we describe the applications and cloud settings we consider, as well as the algorithm parameters we use in Section VI-B. The results themselves are explored in detail in Sections VI-C and VI-D.

A. Experiment setting

We test the MALIBOO algorithm on four different scenarios, which act as representatives of different workload types which are relevant to cloud systems. The first three scenarios involve the Apache Spark big data analytics framework [40], while in the fourth one the Stereomatch edge computing application [41] is used. Big data applications are often run on cloud servers because they offer easy access to powerful analysis frameworks such as Apache Spark. On the other hand, an edge computing application such as Stereomatch represents a emerging access pattern to cloud resources, in which data is collected at the IoT layer, but are processed in the cloud, possibly partially.

We now describe the three Spark applications, which are being optimized on the number x of parallel cores used to run them (x is therefore one-dimensional):

- *Query26* is an interactive query from the TPC-DS industry benchmark[1], and represents SQL-like tasks. We execute it with input datasets I of varying size, specifically 250 GB, 750 GB, and 1000 GB.
- *Kmeans* is a well-known ML clustering technique, and a typical example of an iterative task. We execute it on Spark-Bench[2] by providing it as input datasets I with 100 features (columns) and varying size (rows): 5, 10, 15, and 20 million.
- *SparkDL Transfer Learning*[3] is a big data analytic tool which applies transfer learning to Deep Learning applications, through the ML Pipelines from Spark MLlib. In this paper, we consider an image binary classification task with input datasets I containing 1000, 1500, and 2500 images.

Spark experiments are conducted on two different computing environments: the Microsoft Azure public cloud and a private IBM Power8 cluster. In particular, Query26 and SparkDL are executed on Microsoft Azure using the HDInsight service with workers based on 6 D13v2 virtual machines (VMs), each with 8 CPU cores and 56 GB of memory. Instead, Kmeans is run on an IBM Power8 deployment that includes 4 VMs, each with 12 cores and 58 GB of RAM, for a total of 48 CPU cores available for Spark workers, plus a master node with 4 cores and 48 GB of RAM. These two systems are representatives of different computing environments. The Microsoft Azure public cloud is potentially affected by resource contention, and application execution times might experience more variability as a result. Whereas, the private IBM Power8 cluster is fully dedicated to our experiments without any other concurrent activity, i.e., with no resource contention.

For all three Spark applications, we perform one separate experiment for each input dataset I, for a total of 10 experiments. Moreover, for each of these applications, we also carry out one extrapolation experiment. In particular, we fix the largest input dataset I available for each application (1000 GB, 20 million rows, and 2500 images respectively), but we give additional profiling data to the ML model $\widehat{T}(\cdot)$ estimating the performance (see Section V-A). These additional profiling data consist of all previous runs with smaller input datasets I (250-750 GB, 5-10-15 million rows, and 1000-1500 images respectively) and are used in the training phase of the regression model $\widehat{T}(\cdot)$, in addition to the points in history H collected by BO (see Algorithm 2, step 6). In fact, in a big data setting, it is often required to run data analysis tasks or prediction models on datasets with increasing size. Moreover, if the extrapolation model is accurate on large input datasets, when using smaller input datasets, this allows to save on exploratory runs with the larger datasets, since the latter are usually much more expensive. We also note that, according to preliminary analysis in [12], ML models show good extrapolation capabilities on the applications of interest.

The fourth scenario we consider for validation uses *Stereomatch* [41], an image-processing edge computing application that computes the disparity value between a pair of stereo images (i.e., coming from the same scene but observed by two cameras), which can then be used to calculate the depth

[1]http://www.tpc.org/tpcds

[2]https://codait.github.io/spark-bench

[3]https://github.com/databricks/spark-deep-learning

of objects in that scene. This application uses adaptive-shape local support windows for each pixel, based on color similarity. In this case, x consists of four independent parameters which influence its execution time: number of parallel threads, color similarity confidence, granularity of the disparity hypotheses to be tested, and length of the arm of the support windows. This is a much larger parameter space than the other experiments, in which we have a mono-dimensional optimization problem on the number of cores only. This application is executed with a fixed input dataset I containing 40 pairs of images, on a Virtual Machine (VM) on a private 32-core server with 64 GB of memory and Ubuntu 20.04. The underlying physical node of this server has two AMD EPYC 7282 processors, with 16 cores and 32 threads, and clock speed 2.8 GHz. Since we use a single input dataset for Stereomatch, we perform a single experiment with it. The total number of experiments we perform across all four scenarios is therefore 14, including the three extrapolation experiments.

We run MALIBOO from an Ubuntu 20.04 machine with 16 GB RAM for 30 maximum iterations for each experiment, or 60 in the Stereomatch scenario because of the larger search space. Besides the execution time of the job, the computation time for a single algorithm iteration is about 1.5 seconds, with a maximum of 5 seconds for late iterations (in which larger ML models are being trained), and up to 10 seconds for extrapolation experiments.

B. Algorithm settings

We compare the four algorithm variants (A-D, see Section V-A) in all 14 experiments concerning the four applications described in the previous section. For each experiment, we use the same initial configurations and time thresholds. In particular, for each experiment, we choose $n_0 = 3$ initial configurations and repeat the experiments with different time thresholds T_{max}. Specifically, for each experiment, we choose a grid of 10 evenly spaced thresholds. This represents an increasingly difficult optimization problem as the time threshold gets smaller, since the feasible domain keeps shrinking.

We use a Ridge linear regression model as the predicting function $\widehat{T}(\cdot)$ described in Section V-A. We also give the model some additional features derived from the ones in x, namely the inverse of the number of parallel cores, its logarithm (which encodes the cost of reducing operations in parallel frameworks, see [42]), and the pairwise products of all involved features. This ML model is computationally cheap, and performs better in predicting the performance of the applications of interest when compared to other alternative methods (see the analysis reported in [12]). In particular, we measure its predicting accuracy by computing the Mean Absolute Percentage Error (MAPE) of the model:

$$MAPE(\boldsymbol{y}, \widehat{\boldsymbol{y}}) = \frac{1}{N} \sum_{i=1}^{N} \left| \frac{y_i - \widehat{y}_i}{y_i} \right|,$$

where $\boldsymbol{y}$ is the vector of true values and $\widehat{\boldsymbol{y}}$ is the vector of predicted values by the ML model. According to our analysis on the full profiling datasets, the test-set MAPE of the chosen model is almost always lower than 9%. Even when just a small handful of training points is available, errors are mostly within 20%. As for the GP hyperparameters, we choose a constant mean function and a Matérn kernel, as described in Section IV, with smoothness parameter $\nu = 5/2$. We choose $k = 2$ as the exponential term used in variants B and D (see Section V-A), $q = 10$ as the length of the memory queue, and $\alpha = 0.9$ as the lower bound parameter for the stopping criterion (see Section V-B).

We now compare our proposed algorithm variants (B-D) with variant A, which uses the same acquisition function as *CherryPick* [1] (see also Section IV-B), in terms of number of unfeasible runs and their cumulative costs. After that, we will describe the results of the extrapolation analysis (see Section VI-D), in which we feed the ML models with profiling data with smaller input datasets. For each of these scenarios, we also show the percentage error of the ML models.

C. Numerical results

The first four rows of Table I summarize the average results on the applications described in Section VI-A, over the varying input data sizes and time thresholds. In particular, we report: the number of executions which selected an unfeasible configuration; the percentage of costs (the $f(x)$ in Eq. (1)) coming from unfeasible configurations over the total amount of costs; and the average cost among feasible configurations, normalized over the cost of variant A. For

TABLE I
MEASURED METRICS FOR VARIANTS A-D OF THE PROPOSED ALGORITHM

scenario	var. A	var. B	var. C	var. D
Query26				
unfeasible runs	11.23	4.34	4.11	4.31
ratio of unfeasible costs	0.36	0.14	0.13	0.14
mean feasible cost	1.00	0.87	0.87	0.88
Kmeans				
unfeasible runs	10.52	5.74	5.93	5.96
ratio of unfeasible costs	0.34	0.20	0.21	0.21
mean feasible cost	1.00	1.00	1.01	1.00
SparkDL				
unfeasible runs	11.65	5.29	5.14	5.24
ratio of unfeasible costs	0.32	0.16	0.15	0.16
mean feasible cost	1.00	0.83	0.87	0.79
Stereomatch				
unfeasible runs	18.87	13.83	10.83	8.70
ratio of unfeasible costs	0.27	0.22	0.17	0.15
mean feasible cost	1.00	0.78	0.86	0.69
Query26-extrapolation				
unfeasible runs	3.57	4.64	4.64	4.86
ratio of unfeasible costs	0.11	0.15	0.15	0.15
mean feasible cost	1.00	0.83	0.83	0.83
Kmeans-extrapolation				
unfeasible runs	10.00	5.00	5.00	5.00
ratio of unfeasible costs	0.33	0.17	0.17	0.17
mean feasible cost	1.00	1.33	1.40	1.32
SparkDL-extrapolation				
unfeasible runs	12.75	8.62	6.29	5.57
ratio of unfeasible costs	0.35	0.27	0.19	0.16
mean feasible cost	1.00	0.98	0.83	1.13

the Query26 application, our algorithm variants reduce the unfeasible executions and the ratio of unfeasible costs 2 to 3 times with respect to pure BO (variant A). The average cost of a feasible configuration is also reduced. On average, in the three Spark applications we see an average improvement of 2.2 times on the number of unfeasible runs and 2 times on the total unfeasible cost. The cost of feasible runs improves by 10% on average, and it is always at least competitive with variant A in the worst case. As for the multi-dimensional case with the Stereomatch application, results are in line with the other three scenarios, with the average improvement on feasible costs further improving to about 25%.

A representative example for a run of our proposed algorithm is displayed in Fig. 2, which represents the Query26 scenario. In the left panel, we represent the number of cores

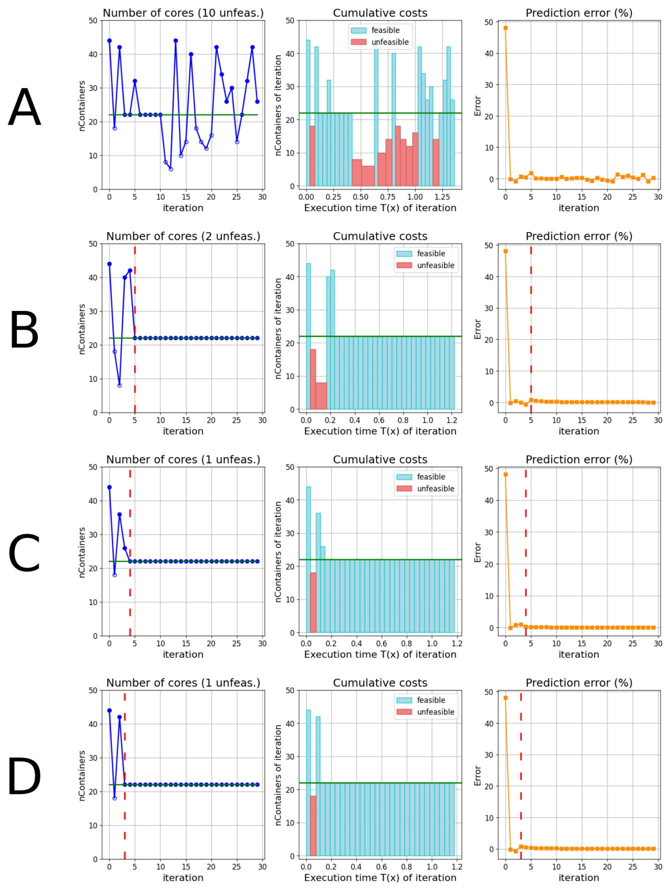

Fig. 2. Query26: comparison of pure BO (variant A) with variants B-D.

chosen at each algorithm iteration. The green horizontal line is the true optimum of the constrained optimization problem, which we identified by inspection through an exhaustive profiling of the application. The vertical, dashed red line indicates the run at which the stopping criterion kicks in, and after which we stick to the best configuration found so far by the algorithm. In the center panel, a rectangle represents a single run, with its sides being the execution time of the job (horizontal side) and the number of cores (vertical side). Therefore, the area of a rectangle is proportional to the execution cost $f(x)$ for that particular run (see Eq. (1)). We highlight in red the rectangles corresponding to unfeasible configurations. Finally, the signed percentage errors of the ML model for the execution time are displayed in the right panel. In particular, at each iteration, the model is trained with all data from previous iterations (i.e., ones in the history H), and the error is evaluated on the new configuration chosen by the algorithm. We see that after one initial run with large error, our ML models quickly converge to errors very close to zero. The spike at the first iteration is explained by the fact that the BO algorithm is still in the exploration phase, and selects a configuration with a large number of cores since that region of the domain is still unexplored. This also means that the ML model struggles in the first iteration, since it was trained with data with small amounts of cores. After that, the ML model has accumulated enough information to achieve good predicting capabilities, even with a small amount of training data points. As was shown in Table I, from Fig. 2, it is clear that our algorithm drastically reduces the number of unfeasible runs (from 10 to only 1-2) while still achieving the optimum. We show a similar plot for the Kmeans application in Fig. 3. Note that, despite the ML model being less accurate on

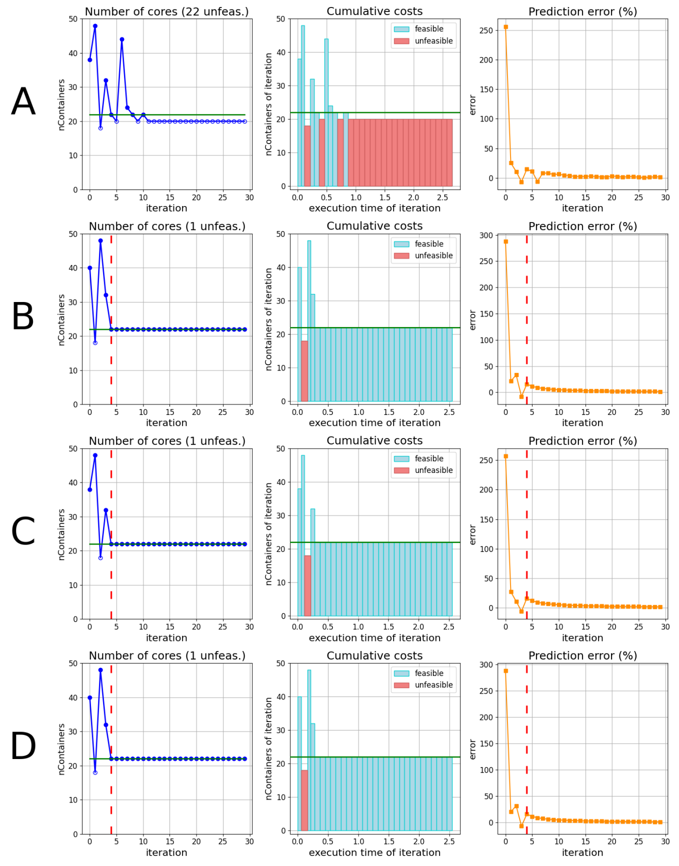

Fig. 3. Kmeans: comparison of pure BO (variant A) with variants B-D.

the Kmeans application when compared to other workloads (about 20% MAPE error on before convergence, compared to less than 9% for Query26), results are still in line with the other cases. Overall, our algorithm variants converge to an optimal or feasible near-optimal (i.e., one more or one less than the optimum) number of cores within the given iteration budget over 54% of the times, usually before the 10th execution. Variant A (i.e., pure constrained BO) achieves a similar percentage in the Query26 and Kmeans experiments, but fails to converge to a feasible near-optimal configuration in all experiments involving the SparkDL and Stereomatch applications.

D. Extrapolation runs with additional data

We also perform extrapolation experiments with the three Apache Spark applications: Query26, Kmeans, and SparkDL. We recall that these experiments correspond to the usual need to run the same big data analysis on datasets of increasing size, and are performed by providing additional profiling data from smaller datasets to the ML model in the training phase (see Section VI-A).
A representative extrapolation result is shown on the right side of Fig. 4, while the left side reports the corresponding case with "vanilla" algorithms, without additional training data being fed to the ML models. (Variant A is identical on both sides, since vanilla BO does not use any ML model.) One can immediately notice the lack of the initial spike in the error diagrams of variants B-D, owing to the fact that the ML model is trained with more data points from the very beginning. Thanks to the aforementioned good extrapolation capabilities (see [12] for additional discussion), the ML model is therefore able to accurately predict the execution time, even when it is presented with a point from a region that the BO algorithm has not explored yet. Results of all runs averaged on the varying time thresholds are collected in the lower part of Table I.

In general, the algorithm with additional data has a more aggressive search behavior, trading off a small amount of unfeasible runs (2 at most) for a lower average cost of individual runs (6 to 26%). This is explained by the additional data making the trained ML model more accurate: the model guides the domain exploration of BO towards more promising points, either because of their lower predicted execution time (variant B), their predicted feasibility (variant C), or both (variant D). These points are thus more likely to be on the boundary of the feasibility region of the optimization problem, where the value of the objective function is close to the optimum, but also where we have a higher risk of slipping out of the feasibility region. The extrapolation version of MALIBOO is therefore best suited for the user with a lower risk aversion.

In conclusion, each of our algorithm variants (B-D) outperforms the constrained vanilla BO technique in [1] (here represented by variant A) with respect to the number of unfeasible runs and the total amount of unfeasible costs (2 to 3 times less), as well as the average cost of the unfeasible configuration (13 to 22% less).

VII. Conclusions and future work

In this paper, we have presented MALIBOO, a tool combining a BO algorithm with ML techniques, to find the optimal configuration for a recurring job running on a cloud server, under a given time threshold. Results on tested edge computing and big data analysis applications show that our algorithm significantly reduces the amount of unfeasible executions with respect to a pure BO approach, as well as reducing the average cost of the configuration. Overall, each of our algorithm variants outperforms the state-of-the-art BO technique used as benchmark.

We plan on testing the hybrid algorithm implemented by MALIBOO on other application domains, e.g., High-Performance Computing systems and AI applications, while also considering acquisition functions suitable for a parallel exploration of alternative configurations.

References

[1] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI Proc.*, 2017.

[2] L. Wang, L. Yang, Y. Yu, W. Wang, B. Li, X. Sun, J. He, and L. Zhang, "Morphling: Fast, near-optimal auto-configuration for cloud-native model serving," in *ACM SoCC Proc.*, 2021.

[3] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies, "Arrow: Low-level augmented Bayesian optimization for finding the best cloud vm," in *ICDCS Proc.*, 2018.

[4] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *ICDM Proc.*, 2016.

[5] P. I. Frazier, "A tutorial on Bayesian optimization," *arXiv:1807.02811 preprint*, 2018.

[6] T. Pourmohamad and H. K. Lee, "Bayesian optimization via barrier functions," *Journal of Computational and Graphical Statistics*, vol. 31, no. 1, pp. 74–83, 2022.

[7] M. Schonlau, W. J. Welch, and D. R. Jones, "Global versus local search in constrained optimization of computer models," *Lecture Notes-Monograph Series*, pp. 11–25, 1998.

[8] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *IEEE Proc.*, vol. 104, no. 1, 2015.

[9] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *SIGKDD Proc.*, 2017.

[10] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," *NIPS Proc.*, 2012.

[11] M. Lattuada, E. Gianniti, D. Ardagna, and L. Zhang, "Performance prediction of deep learning applications training in gpu as a service systems," *Cluster Computing*, vol. 25, no. 2, pp. 1279–1302, 2022.

[12] A. Maros, F. Murai, A. P. C. da Silva, J. M. Almeida, M. Lattuada, E. Gianniti, M. Hosseini, and D. Ardagna, "Machine learning for performance prediction of spark cloud applications," in *IEEE CLOUD Proc.*, 2019.

[13] P. Nawrocki and P. Osypanka, "Cloud resource demand prediction using machine learning in the context of qos parameters," *Journal of Grid Computing*, vol. 19, no. 2, pp. 1–20, 2021.

[14] D. F. Kirchoff, M. Xavier, J. Mastella, and C. A. De Rose, "A preliminary study of machine learning workload prediction techniques for cloud applications," in *Euromicro PDP Proc.*, 2019.

[15] S. Mustafa, I. Elghandour, and M. A. Ismail, "A machine learning approach for predicting execution time of spark jobs," *Alexandria engineering journal*, vol. 57, no. 4, pp. 3767–3778, 2018.

[16] X. Pan, S. Venkataraman, Z. Tai, and J. Gonzalez, "Hemingway: Modeling distributed optimization algorithms," *NIPS Proc.*, 2016.

[17] A. Y. Nikravesh, S. A. Ajila, and C.-H. Lung, "Towards an autonomic auto-scaling prediction system for cloud resource provisioning," in *SEAMS Proc.*, 2015.

[18] Q. Li, B. Li, P. Mercati, R. Illikkal, C. Tai, M. Kishinevsky, and C. Kozyrakis, "Rambo: Resource allocation for microservices using Bayesian optimization," *IEEE Computer Architecture Letters*, vol. 20, no. 1, pp. 46–49, 2021.

[19] K. Kawaguchi, L. P. Kaelbling, and T. Lozano-Pérez, "Bayesian optimization with exponential convergence," *NIPS Proc.*, 2015.

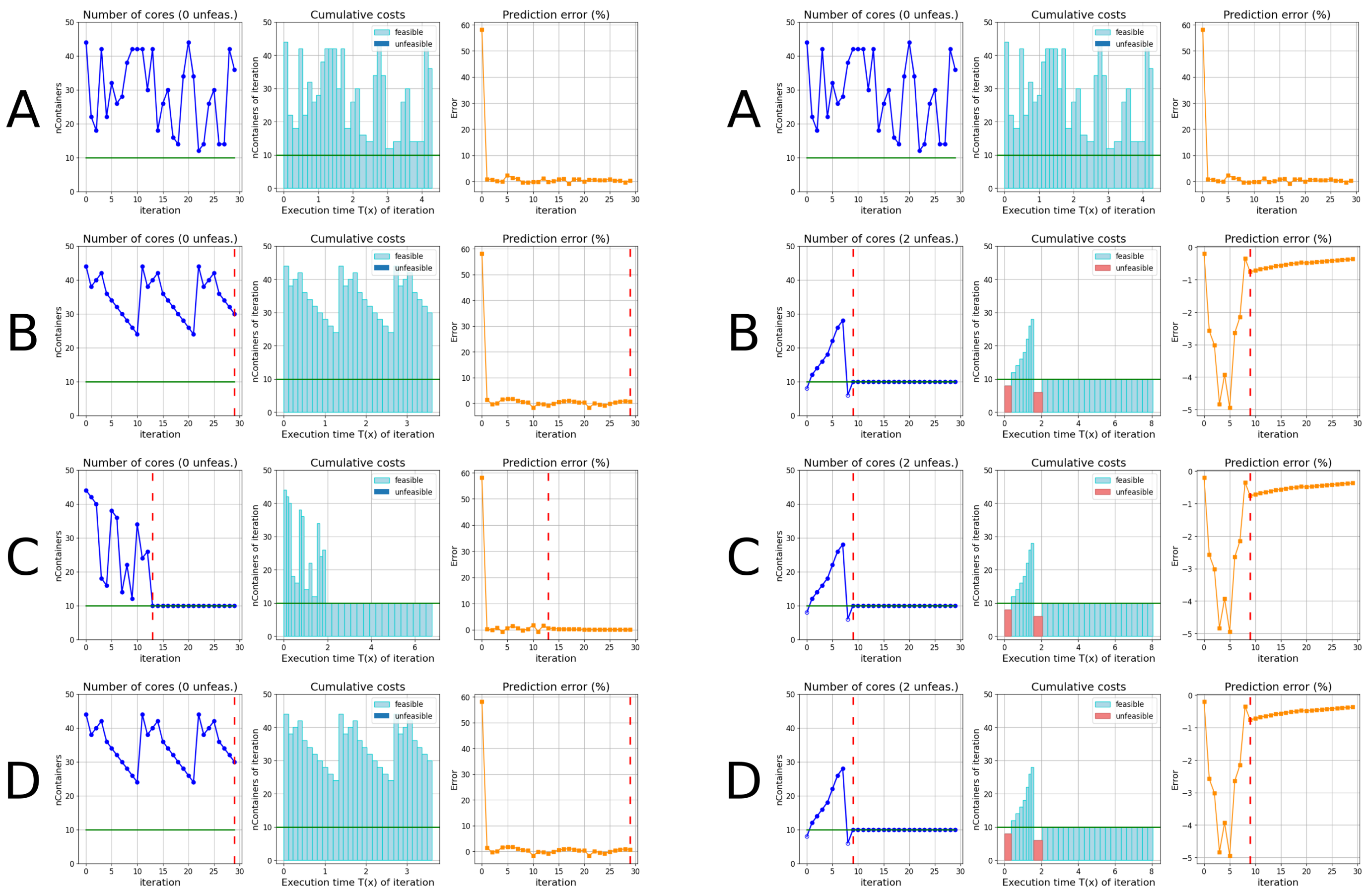

Fig. 4. Comparison between runs without additional data (left) and ones with additional extrapolation data (right).

[20] T. S. Frisby, Z. Gong, and C. J. Langmead, “Asynchronous parallel Bayesian optimization for ai-driven cloud laboratories,” *Bioinformatics*, vol. 37, no. Supplement_1, pp. i451–i459, 2021.

[21] L. Fang, E. Makkonen, M. Todorovic, P. Rinke, and X. Chen, “Efficient amino acid conformer search with Bayesian optimization,” *Journal of chemical theory and computation*, vol. 17, no. 3, pp. 1955–1966, 2021.

[22] S. Jalas, M. Kirchen, P. Messner, P. Winkler, L. Hübner, J. Dirkwinkel, M. Schnepp, R. Lehe, and A. R. Maier, “Bayesian optimization of a laser-plasma accelerator,” *Physical review letters*, vol. 126, no. 10, p. 104801, 2021.

[23] S. Ariafar, J. Coll-Font, D. H. Brooks, and J. G. Dy, “Admmbo: Bayesian optimization with unknown constraints using admm,” *J. Mach. Learn. Res.*, vol. 20, no. 123, pp. 1–26, 2019.

[24] B. Letham, B. Karrer, G. Ottoni, and E. Bakshy, “Constrained Bayesian optimization with noisy experiments,” *Bayesian Analysis*, vol. 14, pp. 495–519, 2019.

[25] M. A. Gelbart, J. Snoek, and R. P. Adams, “Bayesian optimization with unknown constraints,” *UAI Proc.*, 2014.

[26] A. Das, A. Leaf, C. A. Varela, and S. Patterson, “Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications,” in *IEEE CLOUD Proc.*, 2020.

[27] Y. Ding, A. Pervaiz, S. Krishnan, and H. Hoffmann, “Bayesian learning for hardware and software configuration co-optimization,” *TR-2020-13 University of Chicago*, 2020.

[28] A. Candelieri and F. Archetti, “Sequential model based optimization with black-box constraints: Feasibility determination via machine learning,” in *AIP Proc.*, vol. 2070, no. 1, 2019.

[29] R. Perego, A. Candelieri, F. Archetti, and D. Pau, “Tuning deep neural network’s hyperparameters constrained to deployability on tiny systems,” in *ICANN Proc.* Springer, 2020.

[30] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, W. Zwaenepoel, and D. Garlan, “Lynceus: Cost-efficient tuning and provisioning of data analytic jobs,” in *ICDCS Proc.*, 2020.

[31] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, “Reoptimizing data parallel computing,” in *NSDI Proc.*, 2012.

[32] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, “Jockey: guaranteed job latency in data parallel clusters,” in *EuroSys Proc.*, 2012.

[33] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini, “History-based harvesting of spare cycles and storage in large-scale datacenters,” in *OSDI Proc.*, 2016.

[34] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*. MIT press Cambridge, MA, 2006, vol. 2, no. 3.

[35] M. Abramowitz, “Handbook of mathematical functions,” *US Department of Commerce*, vol. 10, p. 375, 1972.

[36] D. R. Jones, M. Schonlau, and W. J. Welch, “Efficient global optimization of expensive black-box functions,” *Journal of Global optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[37] P. Hennig and C. J. Schuler, “Entropy search for information-efficient global optimization,” *JMLR*, vol. 13, no. 6, 2012.

[38] X. Luo, “Minima distribution for global optimization,” *arXiv:1812.03457 preprint*, 2018.

[39] D. Cvijović and J. Klinowski, “Taboo search: An approach to the multiple-minima problem for continuous functions,” in *Handbook of global optimization*. Springer, 2002, pp. 387–406.

[40] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, “Apache spark: a unified engine for big data processing,” *Commun. ACM*, vol. 59, no. 11, 2016.

[41] E. Paone, G. Palermo, V. Zaccaria, C. Silvano, D. Melpignano, G. Haugou, and T. Lepley, “An exploration methodology for a customizable opencl stereo-matching application targeted to an industrial multi-cluster architecture,” in *CODES Proc.*, 2012.

[42] P. Pacheco, *An introduction to parallel programming*. Elsevier, 2011.